郑州大学毕业设计（论文）

题 目： 同态加密硬件加速器模拟系统设计

指导教师： 范文兵 职称： 教授

指导教师(校外)： 赵路坦 职称： 博士后

学生姓名： 蔡明 学号： 201924110803

专　业： 电子信息工程

院（系）： 电气与信息工程学院

完成时间： 2022 年 5 月 19 日

2022 年 5 月 19

摘要

全同态加密（Fully Homomorphic Encryption, FHE）技术保护了数据安全，但是，FHE 的计算开销是明文直接计算开销的上万倍。因此，为了弥补性能上的巨大差距，最近越来越多的基于 GPU、ASIC 和 FPGA 的硬件加速器开始出现。但是在硬件加速器设计过程中，如何选择合适的加速器架构参数是很困难的，因此加速器的设计过程中往往需要软件模拟器的配合。本文通过使用 python 语言进行模拟加速器架构，设计了一款基于同态加密 FPGA 硬件加速器的软件模拟器，该模拟器可以辅助架构设计人员调整加速器架构设计中的一些参数，比如外部存储内存带宽、加速器输入缓存和输出缓存的大小以及加速器片上带宽等等。该模拟器通过模拟执行一组同态操作，可以得到参数与执行这组同态操作所需总周期数的关系。最后在指定场景下通过设计实验和分析模拟器的输出报告成功得到了加速器架构的局部最优参数。

关键词 全同态加密，加速器，模拟器，FPGA

Abstract

Though Full Homomorphic Encryption (FHE) technology protects the data security, the computing cost of FHE is tens of thousands of times that of plaintext direct computing. Therefore, in order to make up for the huge gap in performance, more and more hardware accelerators based on GPU, ASIC and FPGA have emerged recently. However, it is difficult to select appropriate architecture parameters in the process of accelerator design. So, the accelerator design process often needs the cooperation of software simulator. I design a software simulator based on FPGA hardware accelerator by using python language to simulate the accelerator architecture. This simulator can help architecture designers adjust some parameters in the accelerator architecture design, such as external storage memory bandwidth, the size of accelerator input cache and output cache, on-chip bandwidth. The simulator simulates the execution of a set of homomorphic operations to obtain the relationship between parameters and the total number of cycles required to perform this set of homomorphic operations. Finally, the local optimal parameters of the accelerator architecture were successfully obtained through designing experiments and analyzing the output report of the simulator in the specified scenario.

Key Words fully homomorphic encryption, accelerator, simulator, FPGA

目 录

1 绪论

1.1 研究背景和意义

同态加密可以在保持数据的加密的同时进行计算，这颠覆了传统加密方式无法加密计算的局面。同态加密可以广泛应用于云计算、金融科技、多方安全计算和健康医疗等领域，为保护数据安全提供了安全和高效的解决方案。在云计算领域，同态加密可以解决在云平台上进行计算时数据隐私泄露的问题[1]。同时，同态加密可以减少多方参与计算时数据交换的问题，大大提高计算速度。在金融科技领域，同态加密可以实现保证隐私的交易处理，可以有效防止黑客入侵和数据窃取。在健康医疗领域，同态加密可以保护个人隐私，同时允许研究人员访问高度敏感的健康数据，以促进医学研究的进步。

但是同态加密，尤其是全同态加密的运算会导致计算成本、时间开销达到明文直接计算的上万倍。因此，为了提高 FHE 的效率，加速器的设计是一个不可避免的方向，近年来也出现了越来越多的基于 FPGA、GPU 和 ASIC 的硬件加速器,因为 FPGA 和 GPU 的灵活性以及 ASIC 的高性能，所以三者都有[2][3][4][5]。但是，为了设计出合理高效的硬件架构，在设计过程中往往需要进行繁琐的实验进行架构参数调整，而通过模拟器进行硬件加速器的设计调参可以大大降低研究时间和成本。因此，一种能够辅助研究者进行加速器设计调参的模拟器是非常必要的。

1.2 国内外研究现状

1.2.1 同态加密加速器设计的研究现状

对于同态加密技术的应用场景，如云计算和物联网中，计算速度和保护隐私往往是一对矛盾。在这种情况下，研究人员已经开始探讨使用硬件加速器来提高同态加密算法的计算性能。目前主要的研究成果如下：

2018 年，Axel Feldmann 等人提出了一种名为 F1 的计算加速器[3]，用于支持全同态加密（FHE）技术下的安全高效计算。该加速器是使用 Verilog HDL 描述并实现，在 Xilinx Virtex-7 FPGA 平台上进行了测试。

2020 年，Nikola Samardzic 等人基于 Axel Feldmann 等人提出的 F1 加速器，结合 FHE 的密文大小通常非常大这一背景对加速器进行了改进，提出了 CraterLake 加速器[6]。

此外还有 Yinghao Yang 等人提出的 Poseidon 加速器[7]、Rashmi Agrawal[8]等人提出的 FAB 加速器以及 Soran Ahmed[9]等人提出的一种针对 RSU 资源受限网络中全同态加密应用和安全性需求特点的加速器设计，实现了 FHE 算法的低功耗和高效计算。

1.2.2 硬件加速器模拟器设计的研究现状

硬件加速器模拟器是一个用于模拟硬件加速器的软件工具，它可以在不需要专用硬件平台的情况下，评估加速器的性能、功耗等指标。模拟器又分为系统级模拟器、功能级模拟器和行为级模拟器，本文主要考虑功能级模拟器。在针对功能级模拟器设计方面目前有:

Ananda Samajdar 等人针对 DNN 卷积神经网络脉动阵列加速器设计的周期精准的 SCALE-SIM 模拟器[10]。SCALE-Sim 是一个专为深度学习应用设计的模拟器，可以支持各种常见的 DNN 架构（如 CNN、RNN、LSTM 等）及其优化技术。

Daniel J.Sorin 等人开发用于内存模拟的 DRAM-Sim2[11]，它主要关注模拟 DRAM 中存储单元和数组的操作，以及内存控制器的操作。通过模拟内存的具体操作行为，DRAM-Sim2 可以帮助开发者进行内存系统的性能分析和优化。相比于功能级模拟器，行为级模拟器更加逼真，能够更好地反映实际系统的行为。

Binkert M.I 等人出的用于通用处理器的 GEM5 模拟器[12] 。通常被用于 CPU 架构相关的模拟器和仿真领域。它支持多种 ISA（Instruction Set Architecture）CPU 体系结构（如 ARM、x86、MIPS 等），可以用于研究微处理器和计算机系统的各种性能指标、缓存一致性、功耗以及体系结构调优等方面的研究。

目前还没有用于同态加密加速器设计的专用模拟器。

1.3 本文主要工作

本文主要是基于现有的全同态加密加速器架构设计一种用于辅助调整同态加密加速器架构设计参数的行为级周期精准模拟器。

基于同态加密算法本身影响加速器架构，加速器架构影响模拟器设计的这样一个思路将本文主要工作分为以下内容：

（1）对同态加密算法的基本概念、应用场景以及各种同态算法进行了基本介绍，然后针对本文采用的 CKKS（Cheon-Kim-Kim-Song）全同态算法进行了算法原理的分析以便后文的加速器设计和模拟器设计的阐述更加清晰。

（2）对现有的基于 ASCI 和 FPGA 的同态加密硬件加速器的架构进行了分析，根据其架构特点指出一种用于架构设计的模拟器的必要性和实用性，并详细介绍了本文所模拟的加速器架构。

（3）对基于 python 设计的模拟器的软件系统框图、各模块子框图以及所采用的的数据结构、功能函数等进行了介绍。

（4）介绍如何通过设计好的模拟器进行设计实验来分析加速器架构参数和加速器性能关系，并对一假定场景进行了实验设计和结果分析。

2 同态加密介绍和理论分析

2.1 同态加密算法概述

2.1.1 同态加密算法的介绍

在密码学领域中，使用同态作为一种加密类型。同态加密（HE）是一种加密方案，它允许云服务商这样的第三方对加密的数据执行某些可计算的功能，同时保留加密数据的功能和格式的特征。第三方在得到同态加密的数据之后，对密文进行特定的计算，得到的计算结果再进行同态解密就可以得到和使用明文直接进行计算相同的结果[13]。

典型的同态操作有加法同态和乘法同态，如图 2.1 所示，假设有两条明文消息 m1 和 m2，则有：

对于同态加法，可以满足等式（2.1）和（2.2），其中 Dec 是同态解密函数、Enc 是同态加密函数[14]。

$$Dec(\ Enc\ (\ m1\ ) + Enc\ (\ m2\ )\ ) = m1 + m2 \tag{2.1}$$

$$Enc(m1) + Enc(m2) = Enc(m1 + m2) \tag{2.2}$$

对于同态乘法，可以满足等式（2.3）和（2.4），其中 Dec 是同态解密函数、Enc 是同态加密函数。 对于 HE（Homomorphic Encryption）中的表示法，它只允许加法和乘法运算。对于任何布尔电路，只能通过 XOR 门执行加法和和 And 门执行乘法[16]。

$$Dec\big(Enc(m1) * Enc(m2)\big) = m1 * m2 \tag{2.3}$$

$$Enc(m1) * Enc(m2) = Enc(m1 * m2) \tag{2.4}$$

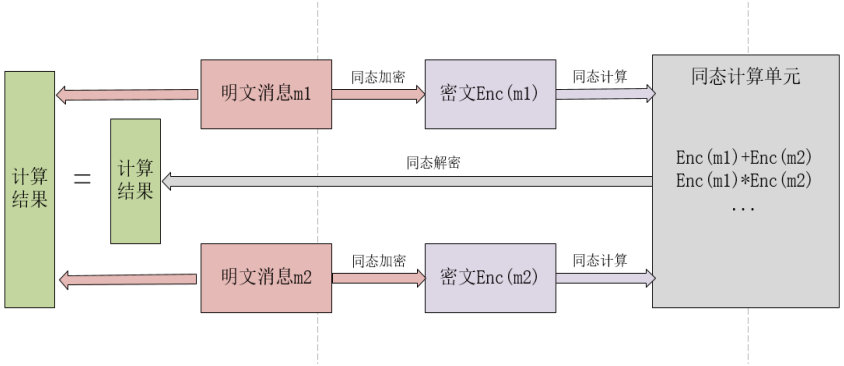

图 2.1 同态加密计算示意图

2.1.1 同态加密算法的分类

在同态加密技术中，为了能够实现在加密状态下进行计算，需要将明文经

过加密算法转换成密文，同时添加一个随机噪声。当使用同态加法或者同态乘法对密文进行运算时，这个随机噪声会随着运算而不断累加，导致密文中的噪声变得越来越大。

当密文中的噪声超过一定的阈值后，就会影响计算结果的准确性，从而导致解密结果的错误。一般情况下，同态加密算法会规定一个最大的可以容忍的噪声上限，如果噪声超过这个上限，则需要重新生成密文或者采用其他技术处理。因此，同态加密算法并不是都可以进行无数次的同态加法或者同态乘法的，同态加密算法可以根据对允许操作的数量的加密数据整齐地归类为三种类型的方案：

PHE（Partially Homomorphic Encryption）部分同态加密，如 Paillier 加密算法 EIGamal 加密算法和 RSA（Rivest Shamir Adleman）加密算法[14]，由于其设计的简单性，可以运行在资源有限的设备上，并且不需要过多的密文噪声控制。因此，PHE 通常拥有很快的运行速度和较低的计算成本，但其功能受到了一定程度的限制。下面对典型 PHE 算法进行简单的介绍：

Paillier 加密算法，该算法的分析主要包括密钥生成、加密、解密和同态操作实现。

- 密钥生成：

  步骤 1：随机的选择两个质数 p 和 q，使其满足等式(2.5)。

$$gcd\big(pq,(p-1)(q-1)\big)=1 \tag{2.5}$$

  步骤 2：通过公式(2.6)和公式(2.7)计算得到 n 和 L。其中 lcm（a,b）是指获得 a 和 b 的最小公倍数。

$$n = p * q \tag{2.6}$$
$$L = lcm(p-1, q-1) \tag{2.7}$$

  步骤 3：随机选择一个满足等式（2.8）的 g，等式（2.8）中的函数 T 如公式（2.9）所示。

$$gcd(T(g^{L} mod n^{2}), n) = 1, g \in Z^{*} \tag{2.8}$$
$$\mathrm{T(x)} = \frac{\mathrm{x-1}}{\mathrm{n}} \tag{2.9}$$

  步骤 4：通过公式（2.10）和（2.11）生成公钥 pubk 和私钥 seck。

$$pubk = (n, g) \tag{2.10}$$
$$seck = (p, q) \tag{2.11}$$

- 加密：

  步骤 1：选择一个随机的正整数 m。

  步骤 2：通过公式（2.12）对一个明文 pla 进行加密得到密文 cic。

$$\mathrm{cic} = \mathrm{Enc}(pla) = g^{pla} * m^{r} \bmod n^{2} \tag{2.12}$$

- 解密： 通过公式（2.13）可以将密文 cic 转成明文 pla。

$$pla = Dec(cic) = \frac{T(pla^{L} mod n^{2})}{T(g^{L} mod n^{2})} \tag{2.13}$$

- 同态计算操作： 公式（2.14）表示了 Paillier 加密算法是如何实现对密文cic_1和密文cic_2同态乘法的。

$$\begin{aligned} \mathrm{Enc}(\mathrm{cic}_1) * \mathrm{Enc}(\mathrm{cic}_2) = \left(\mathrm{g}^{\mathrm{pla}_1} * \mathrm{m}_1^{\mathrm{n}} \mathrm{mod} \mathrm{n}^2\right) * \left(\mathrm{g}^{\mathrm{pla}_2} * \mathrm{m}_2^{\mathrm{n}} \mathrm{mod} \mathrm{n}^2\right) = \\ g^{pla_1+pla_2} * (r_1 + r2)^{n} \mathrm{mod} n^2 \end{aligned} \tag{2.14}$$

SWHE（Somewhat Homomorphic Encryption）有限同态加密，如 Gentry-Halevi-Smart(TFHE)算法、Brakerski-Gentry-Vaikuntanathan(BGV)算法和 Van Dijk-Landais Encrypt Scheme(VLLE)算法。其通常支持多个同态运算类型，如同态加法和少量同态乘法。相比于 PHE，SWHE 提供的更大的功能集，可以执行更复杂、更实用的计算任务。由于 SWHE 使用了更复杂的算法并且需要对噪声进行更精细的控制，在效率方面会稍逊于 PHE。

FHE（Fully Homomorphic Encryption）完全同态加密，如 Gentry-Full Homomorphic EncryptionAlgorithm、Brakerski-Gentry-Vaikuntanathan-Smart(FV)算法和 Cheon-Kim-Kim-Song(CKKS)算法，其是同态加密中最强的形式。FHE 支持任何同态运算类型，包括任意多次的同态加法和同态乘法，即便密文也能直接执行相应操作。FHE 相比于前两者具有更高的安全级别和更高的功能性，但由于其使用了更加复杂的算法和更精细的噪声控制，因此在效率方面通常比 SWHE 和 PHE 更慢。

2.2 CKKS 算法的原理

由于 FHE 算法可以进行任意多次的同态加法和同态乘法，所以 FHE 的实用性更高，并且 FHE 的功能性和安全性也更高。此外，FHE 算法中 CKKS 算法由于其采用近似计算，所以它的细节得到了很大的简化，并且计算效率也有很大的提升，所以现在的加速器主要针对的便是 CKKS 算法。本小节将分析本文模拟的加速器所采用的 CKKS 算法的原理[15]。

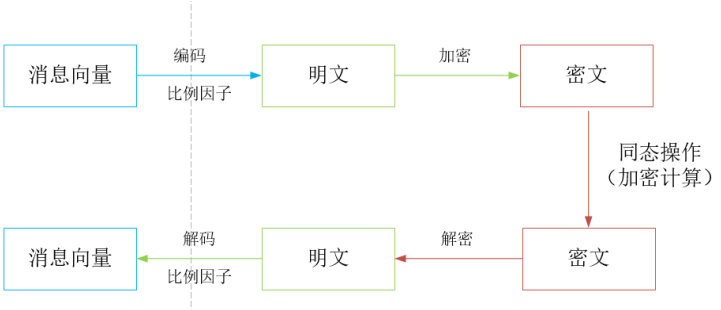

图 2.2 CKKS 计算示意图

如图 2.2 所示，是 CKKS 算法的工作流程，由于 CKKS 算法的处理对象是复数域，所以要先通过编码，将复数消息向量编码成整数多项式，也就是$\mathbb{C}^{\frac{n}{2}} \rightarrow \mathbb{Z}[X]/X^n+1$,然后在进行加密和同态操作，同态计算结果解密之后也需要相应的进行解码变成实际的复数域消息。接下来将逐一介绍 CKKS 算法的编码、解码、密钥生成、加密、解密、同态加法、同态乘法、重线性化、重缩放、旋转、密钥转换（keyswithing）、自举（bootstrapping）以及安全性[17]。

- 编码：完成如（2.15）所示的将消息空间的信息 $Message \in \mathbb{C}^{\frac{n}{2}}$ 编码到明文空间$plaintext \in \mathbb{Z}[X]/X^n+1$。

$$Message \in \mathbb{C}^{\frac{n}{2}} \rightarrow plaintext \in \mathbb{Z}[X]/X^n+1 \tag{2.15}$$

步骤 1：

由于$Message$的长度为$\frac{n}{2}$,并且是复数，所以可以通过将$n/2$个元素取共轭，然后连接到$Message$的后面得到长度为 n 的向量$Message' \in \mathbb{C}^n$。这里之所以要取共轭得到一个长度为 n 的向量，是因为后面需要用到共轭的这个特性得到实数多项式。

步骤 2：

现在我们得到了一个长度 n 的复数向量，其中前$\frac{n}{2}$个元素和后$\frac{n}{2}$个元素是共轭的。我们又知道在复数域内多项式X^n+1有 n 个复数根，并且前$\frac{n}{2}$个复数根和后$\frac{n}{2}$个复数根是共轭的，这 n 个复数根组成一个向量$Root \in \mathbb{C}^n$。

那么便可以通过寻找一个插值多项式$M(x)$，使其满足公式（2.16），其中 Δ 表示的便是图 2.2 中的比例因子，通过它可以将$\mathrm{Message}'$的数值放大。当$Message'$的数值放大之后，再用插值法获得插值多项式，此时的插值多项式的系数相较于没有放大$Message'$就会扩大 Δ 倍，那么取出这个插值多项式的系数的近似整数组成一个整数向量$Plaintext \in \mathbb{Z}(X)/X^n+1$。这个插值多项式可以通过牛顿插值、拉格朗日插值法等方法获得。

$$M(Root_i) \approx \Delta * Message' \tag{2.16}$$

通过上述两个步骤，便可以将消息空间的复数向量编码成整数多项式。

- 解码：解码就是编码的一个逆过程，也就是完成（2.17）所示的转换，将明文空间的整数多项式转成消息空间的复数向量。

$$plaintext \in \mathbb{Z}[X]/X^n+1 \rightarrow Message \in \mathbb{C}^{\frac{n}{2}} \tag{2.17}$$

那么只需要按照编码的相反步骤进行就可以了，如（2.18）(2.19)所示。

$$M_{\Delta^{-1}*plaintext}(Root) \rightarrow Message' \tag{2.18}$$

$$Message'\left[0{:}\frac{n}{2}\right] \rightarrow Message \tag{2.19}$$

- 加密：加密便是把明文加密成密文，现在我们已经有明文整数多项式plaintext，我们要做的事情便是完成公式（2.20）表示的转换。其中$R = \mathbb{Z}[X]/X^n + 1$, Q是一个模数。

$$plaintex \in R \rightarrow cipertext \in R_Q^2, R_Q = \mathbb{Z}_Q[X]/X^n + 1 \tag{2.20}$$

步骤1：

首先要生成加密使用的公钥 pubk 和私钥 seck。为了生成密钥，我们要先取一个离散高斯分布χ_s、一个错误分布χ_e以及一个随机分布χ_t。然后，通过公式（2.21）和（2.22）获得公钥和私钥。

$$seck = (1, \chi_s) \tag{2.21}$$

$$pubk = (-a * \chi_s + \chi_e)\ mod\ Q, a) \tag{2.22}$$

步骤 2：那么通过公钥便可以对明文进行加密得到密文，加密公式如（2.23）所示。

$$cipertext = (ct_0, ct_1) = \chi_t * pubk + (plaintext + \chi_{e0}, \chi_{e1})\ mod\ Q \tag{2.23}$$

- 解密：对于密文cipertext $= (ct_0, ct_1) \in R_Q^2$，我们可以通过公式（2.24）解密得到明文plaintext。

$$plaintext = ct_0 + ct_1 * \chi_s\ (mod\ Q) \tag{2.24}$$

2.3 本章小结

本章首先介绍了同态加密的基本原理，然后对同态加密的三种分类及其典型代表进行了分析，分别是部分同态、有限全同态和完全全同态。然后又对本文接下来将使用的完全同态算法 CKKS 的原理进行了详细的分析，包括 CKKS 算法中的编码、解码、加密、解密、同态加、同态乘、keyswitching、bootstrapping、重线性化、重缩放以及安全性等方面。

3 全同态加密硬件加速器

3.1 现有加速器架构分析

3.1.1 概述

同态加密作为一种重要的保护数据隐私的技术，在安全计算和数据共享领域有着广泛的应用前景。然而，由于其运算复杂度高、效率低等特点，使得同态加密在许多实际场景中很难满足实时性和可扩展性的要求。这时候，使用硬件加速器对同态加密算法进行优化可以大幅提升加密效率和计算速度。

硬件加速器通常是指专门设计的芯片或模块，可以加速计算机任务中的某些繁琐的计算过程。相比于软件实现，硬件加速器更具有并行处理能力和基于自定义指令的加速计算来支持更快、更节能的计算。使用硬件加速器对同态加密进行加速的原理是将同态加密算法的某些计算步骤或核心操作加速到专用硬件上，提高同态加密算法的整体运行效率和计算速度，并显著地减少整个加密过程中需要的处理时间。例如硬件加速器可以通过预先计算表格里面的数值，以及在同态加密中使用的 FFT 变换、NTT 变换、多项式乘除运算、线性代数运算等核心部件使用一些特定硬件模块，使得同态加密计算的速度提升很大程度上。另外，出于安全考虑，有些智能芯片可以内嵌物理随机数发生器、TRNG（真随机数发生器）或 PUF （可靠性不区分）技术等安全模块来保护关键数据和实现高水平的密码保护。

因此，结合专用硬件的同态加密实现能够使机密数据的加密效率有了明显的提升，也为企业的数据安全提供了更强的保障。本小节将对基于 FPGA 和 ASIC 的全同态加密加速器进行分析。

3.1.2 基于 FPGA 的加速器介绍

基于 FPGA 设计全同态加密加速器具有更高的灵活性，因为 FPGA 可以通过重新编程适应不同算法和应用需求，允许运行时配置，可以在开发过程中进行调试和修改。此外，FPGA 加速器可以扩展到具有更多计算资源和更高性能的设备，以满足不断增长的计算需求。并且 FPGA 的设计成本和周期相对于 ASIC 来说要小的多。不过 FPGA 由于受到芯片内交叉连接架构的限制，它的性能可能低于专用 ASIC 设计。尽管大多数 FPGA 并没有完全利用所有可用硬件资源来支持 HE 计算。并且，由于 FPGA 通常使用很多密集的 SRAM 存储器，它们可能会消耗相对较高的功耗。本小节将介绍以下几个基于 FPGA 的全同态加密加速器：F1[3]，并重点介绍 F1 加速器，因为下一小节的 ASIC 加速器便是基于 F1 加速器改进的，而我们的模拟器便是基于下一小节的 ASIC 加速器的架构进行设计的。

（1） F1 加速器

① F1 加速器概述：

F1[3]是一种针对同态加密的高性能硬件加速器。它能够提供不同的计算性能和资源配置来支持流处理、批处理或自适应任务处理等。该加速器可以支持不同类型的同态加密算法，包括 Gentry 的基于理想格的全同态加密方案和 Brakerski Gentry Vaikuntanathan (BGV) 的环同态加密方案。F1 加速器主要由数字信号处理器（DSP）和固定逻辑门阵列（FPGA）构成，其中重点优化了在 FPGAs 中实现同态加密所需的操作 (如乘除运算、加减运算、NTT 变换和多项式插值等)。其设计旨在实现尽可能高的时钟速度，并兼顾硬件面积和功耗参数，从而保障加速器的高性能和低延迟，并为用户提供灵活的编程接口。在 F1 加速器中，同态加密算法的原始输入数据被转化为一组抽象向量并通过专门的电路进行处理。这种转换处理可以有效降低加密过程中产生的数据冗余和计算量，使得同态加密算法的实现更快、更高效。同时，F1 加速器还支持并行计算和异构计算、加密计算与明文计算的同时执行，使得加密速度更快，性能也更稳定。

② F1 的加速主要是通过以下三个方面实现的：

- 对长向量的复杂操作

FHE 使用达到几千个元素长的向量编码信息，并使用模算法对它们进行处理。F1 采用矢量处理，并对 FHE 操作量身定制了宽功能单元，以实现大规模的加速。f1 还采用了专门的 NTT 单元和自同构函数单元。因为数论变换（NTT）和自同构，不是元素级的，需要复杂的数据流，而这些数据很难实现为向量操作。

- 使用超长指令静态调度

FHE 程序是对向量上的算术运算的数据流图。所有操作及其依赖关系都提前知道。F1 通过采用静态调度来利用这一点：在超长指令（Very Long Instruction Word，VLIW）处理器中，所有组件都有固定的延迟，编译器负责调度操作和组件之间的数据移动，没有硬件机制处理冲突的（即没有失速逻辑）。由于这种设计，F1 可以每个周期发出许多操作，最小的控制开销；结合向量处理，F1 可以每个周期发出成千上万个标量操作。

- 减小数据流移动

尽管加速器需要复杂的功能单元，但在目前的技术中，有限的片上存储和内存带宽是大多数 FHE 加速器的瓶颈。因此，F1 主要工作的最小化数据移动。首先，F1 具有显式管理的芯片上内存层次结构，内存结构有很多的 bank 和寄存器堆。第二方面，F1 解耦数据移动，并通过在使用前加载数据来隐藏获取数据的延迟。第三方面，F1 使用 FHE 定制的调度算法来最大化数据重用来充分的利用有限的带宽。第四方面， F1 使用相对较少的具有极高吞吐量的功能单元，而不是像之前的工作那样使用低吞吐量的功能单元。这减少了必须同时驻留在芯片上的数据量，从而允许更高的重用性。

③ F1 加速器所使用的两点已有研究成果：

- 通过 NTT 进行多项式乘法

两个多项式 相乘需要卷积它们的系数，这是一个昂贵的操作。就像用快速傅里叶变换可以更快地实现卷积一样，多项式乘法也可以用数论变换（NTT）实现得更快，这是模算法的离散傅里叶变换的一种变体。NTT 以一个N系数多项式作为输入，并返回一个表示在 NTT 域中的输入的N元素向量。多项式乘法可以在 NTT 域内进行元素级乘法。具体来说，如等式（3.1），其中，$\odot$表示的是元素级乘法。

$$NTT(a * b) = NTT(a) \odot NTT(b) \tag{3.1}$$

因为 NTT 只需要$O(Nlog(N))$次模操作，所以把一个多项式乘法拆分成两个正向 NTT，一个元素间乘法以及一个逆 NTT 就可以做到以$O(Nlog(N))$的复杂度实现多项式乘法。

- 用残差多项式来表示很大的数。

FHE 的密文元素大小很大，可以达到 512 位，但宽的数的算术很昂贵。在我们感兴趣的范围内，模乘数的代价随位宽度呈二次增长。此外，我们需要有效地支持广泛的宽度范围（例如，32 位增量的 64 到 512 位），这既是因为程序需要不同的宽度，也因为模量切换逐步降低系数宽度。残差多项式将一个系数较宽的单个多项式表示为多个系数较窄的多项式。为了实现这一点，我们选择模量Q为L较小的不同素数$Q = q_1 q_2 \cdots q_L$的产物。然后，R_Q中的一个多项式可以表示为L个多项式，分别在$R_{q_1} \ldots, R_{q_L}$中，其中第i个多项式的系数是模q_i的宽系数。

④ F1 的架构图分析：

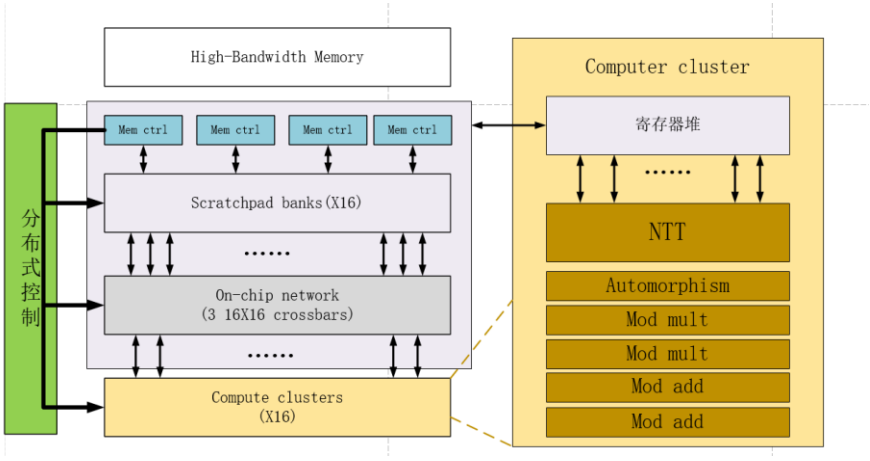

图 3.1 F1 加速器架构图

如图 3.1 所示，F1 加速器的架构由基本向量处理功能单元、计算簇、内存系统、分布式控制以及寄存器堆五部分组成。下面一一对这五部分内容进行介绍：

- 基本向量处理功能单元

F1 的特点是宽向量执行操作与功能单元（FUs）定制的原始 FHE 操作。具体来说，f1 实现了用于模加法、模乘法、ntt 和自同构的向量功能单

元。因为我们利用 RNS 表示，这些功能单元使用固定的、较小的算术字大小（在 F1 实现中是 32 位），避免了大位宽的算术。

- 计算簇

功能单元分组到计算簇中，每个簇都有几个 FUs（1 个 NTT、1 个自同构、 2 个加法器和 2 个加法器）和一个存储寄存器堆，它可以为每个周期提供足够的操作数，使所有 FUs 保持繁忙。该芯片有多个簇（在 F1 的实现中有 16 个）。

- 内存系统

F1 具有一个显式管理的内存层次结构。F1 具有一个大的、大量存储的暂存器。暂存器接口具有高带宽芯片外内存，并通过芯片上网络使用计算簇。F1 使用解耦的数据编排来隐藏主内存延迟。暂存器从主存中获取数据，并且快于它的使用速度。连接暂存器和计算簇的片上网络必须具有非常高的带宽，因为寄存器堆很小，重用能力有限。

- 分布式控制

虽然静态调度是 VLIW 的标志，但 F1 的实现是完全不同的：在 F1 中，并不是每个操作有一个单独的指令流，而是每个组件有一个单独的指令流。虽然 FHE 程序可能有循环，但可以展开它们以避免分支情况，并将程序编译成线性的指令序列。这种方法可能会显得代价高昂。但是向量很长，所以每个指令都编码了大量的工作。此外，还支持一种紧凑的指令格式，表明了相邻两个操作之间的周期数，这种编码提高了 VLIW 指令的低利用率。每个 FU、寄存器堆、网络交换机、暂存器和存储器控制器都有自己的指令流，控制单元从指令 bank 中获取并分配给组件。

- 寄存器堆

F1 中的每个计算簇需要 10 个读端口和 6 个写端口来使所有 FUs 保持工作状态。为了以较低的成本实现这一点，F1 使用了一个 8-bank 分区寄存器文件设计。每个向量分开存储于 bank 之中，每个 FU 随着时间的推移通过所有 bank，每个周期使用一个 bank。通过启动每个向量操作的开始，fu 在每个周期访问不同的 bank。

3.1.3 基于 ASIC 的加速器介绍

基于 ASIC 设计的全同态加密加速器具有更高的性能以及更高的功率效率，并且由于 ASIC 可以严格控制各个计算阶段的物理状态，使其难以收到侧信道攻击等攻击手段，因此其还具有更高的安全性。本小节主要介绍一下 Craterlake 加速器[6]。

Craterlake 是第一个使 FHE 程序具有无界大小（即无界乘法深度）能力的 FHE 加速器。这样的计算需要非常大的密文（每个达到几十个兆的大小）和不同的算法，这些以前的工作都是不支持。为了解决这一挑战，CraterLake 引入了一种新的硬件架构，它可以有效地扩展到非常大的密文，新的功能单元来加

速关键内核，以及新的算法和编译器技术来减少数据移动。CraterLake 在深度 FHE 程序上评估 FHE，包括像 ResNet 和 lstm 这样的深度神经网络，之前的工作在 CPU 上的每个推理需要几分钟到几小时。

先前的 FHE 加速器不能有效地处理无边界深度计算，因为它们支持的是有限大小的向量，并且它们使用的算法在深度程序中很难扩展到大型密文。因此，它们只能运行较小的 FHE 计算，并且它们不支持足够的深度来运行完整的引导过程。Craterlake 是第一个支持无界深度 FHE 计算的 FHE 加速器。为了实现这一点，Craterlake 提供了新的算法、专门的功能单元、硬件架构和编译器技术，以克服深度 FHE 计算的关键挑战：其极端的数据移动需求。

FHE 方案编码信息是一个非常长且每一个元素的位宽都很大的向量。具体地说，支持无界深度计算需要 64K 个元素的向量，每个元素有 1600 位。这使得每个密文需要 25 MB，比之前的 FHE 加速器的目标大 12×。此外，之前的工作采用的 FHE 算法，需要大量的辅助数据。例如，在 F1 中增加 2 MB 的密文需要 32 MB 的辅助数据，而将算法扩展到 25 MB 的密文需要超过 1.4 GB 的辅助数据，这显然不适合存储到芯片上。为了解决这一挑战，Craterlake 采用一种称为增强关键的 FHE 算法，这消除了大部分辅助数据，将开销从 1.4 GB 减少到 50 MB。增强关键化也降低了计算成本。

Craterlake 主要做了以下几点贡献：

- 建立了一种新的宽达 2048 通道的矢量单处理器结构，它将每个向量操作扩展到芯片上。这种矢量单处理器方法减少了并发操作的数量，从而最小化了内存占用空间，减少了芯片外流量，并简化了编译器。
- 为了实现上述的结构，尤其是实现 NTT 和自同构这两个功能，Craterlake 在物理通道的分布组之间建立了一个固定的转置网络。与以前的方法相比，这减少了片上数据的移动和互连成本。
- 一种新的功能单元，可以在运行中生成所需的一半辅助数据（将开销从 50 MB 减少到 25 MB），节省片上存储和内存带宽。
- 一种向量链接技术用来构建长 FU 流水从而用很少的寄存器端口实现许多并发操作。

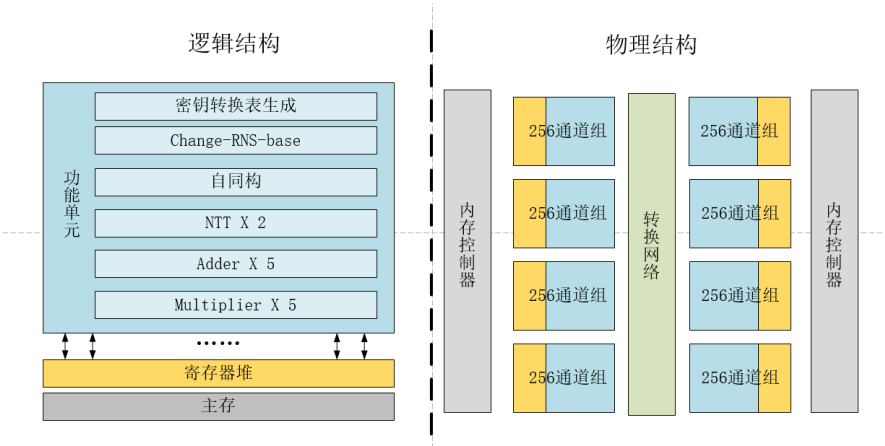

图 3.2 Craterlake 加速器架构

Craterlake 的架构设计如图 3.2 所示，分为逻辑结构和物理结构两部分。

对于逻辑结构组成，主要是功能单元和内存系统。Craterlake 具有专门针对 FHE 操作的功能单元（FUs）。包括从 F1 加速器中引用和改进的模块加法、模乘法、ntt 和快速向量自同构模块。此外，Craterlake 还提供了两个新 的功能单元：一个 Change-RNS-Base 单元（CRB），可以加速大量增强的密钥转换，和一个密钥转换提示生成器，可以在运行中生成每个密钥转换提示的一半，减少了内存流量和片上存储。Craterlake 的片上存储被组织成一个 256 MB 的单级寄存器堆，由所有 FUs 共享。较小的寄存器堆会严重限制性能，该寄存器堆使用一个元素分区的设计来有效地模拟 12 个读和写端口。并且通过允许 fu 被链接起来形成执行更复杂操作的操作。

对于物理结构，实现一个E=2048 通道的向量处理器会导致片上流量过大。Craterlake 通过将其车道分成G=8 车道组来解决这个问题。每个通道组宽为E_G=256，占据芯片物理上不同的区域。由于通道组同时包含 FU 通道和寄存器堆，所以大多数数据移动可以在每个组内本地执行。

3.2 本文所模拟的加速器架构介绍

根据上一小节介绍的 F1 加速器架构和 Craterlake 架构图可以总结出完全同态加密加速器架构的一些要点，首先，加速器是采用静态结构，使用超长向量处理器的形式进行设计的。第二，完全同态加密的每一个密文都是一个向量，而且这个向量很大，在 Craterlake 的场景中甚至能达到几十兆一个密文大小，所以肯定要有一个内存系统，包含大容量高带宽外部存储、以及片上缓存器，我这里将把 Craterlake 中的内存系统进行简化，去掉内存交换网络，并添加一个 FIFO 输出缓存。第三，计算核的每一个功能单元都是全流水的，有固定延迟的，并且没有也不需要控制信号，所以计算核部分可以不用考虑每一个功能单元的具体实现，之余更复杂的操作，也会被分解成原操作，所以在模拟架构设计中也不用考虑。

所以参考 Craterlake 架构图和 F1 加速器架构图以及模拟器的复杂程度，我设计了如图 3.3 所示的这样一个架构图用于后面的模拟器设计。下面将对该架构进行详细的阐述。

如图 3.3 所示，该架构由四部分组成，第一部分是最左边的高带宽内存（High Bandwidth Memory, HBM）,与传统的 DDR 内存相比，HBM 内存具有更高的数据传输速度和更低的能耗。其设计与传统 DDR 结构有所不同，将多个片上存储器封装到一个多层堆叠的 3D 结构中，在同样大小的芯片上可以实现更高的存储密度和更短的访问延迟。此外，HBM 内存还采用了基于片上互连技术的高速互联，从而进一步提升数据吞吐量，并支持与高性能处理器进行更紧密的集成。在需要大量高速内存的领域，如图形/视觉计算、人工智能、数据中心等，HBM 内存已经成为越来越流行的选择。

第二部分是输入缓存，如 3.3 图中中间的上部分所示，整个输入缓存首先被分为多个输入子 buffer，每一个输入子 buffer 的大小是一个密文的最大的大小，也就是 32*65536*60bit 的大小。然后，每一个 buffer 被分为一块一块的 2048bit 的大小的小存储块。这样的输入缓存结构可以减少资源占用，由于每个缓冲区只处理部分数据，因此可以有效减少系统资源占用，从而提高系统

效率。采用多 buffer 结构的主要目的是两个，一个是提高数据的片上重用，第二个是对数据和操作进行解耦，在操作运行之前先把数据存储到 buffer 中，从而隐藏数据读取延迟。

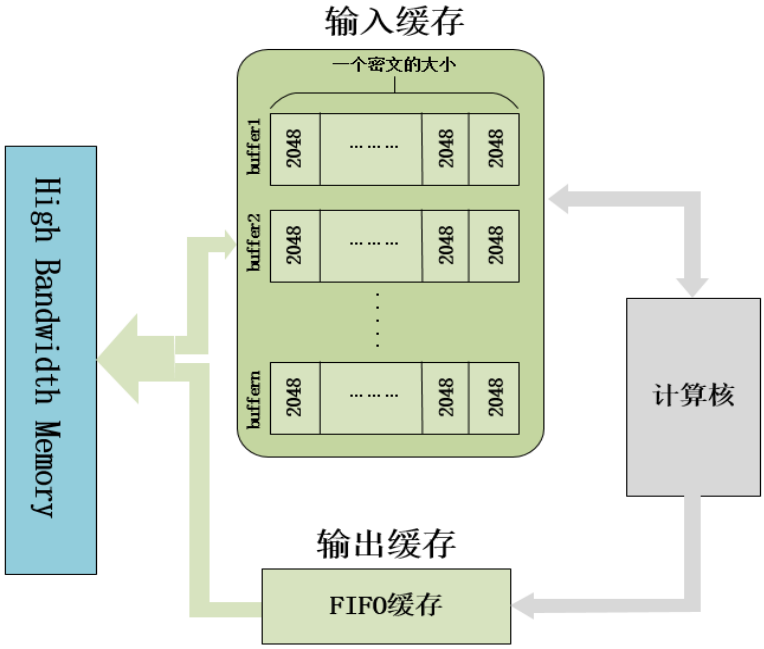

图 3.3 模拟的加速器架构图

第三部分是右边的计算核部分，计算核中是有着固定计算延迟和输入输出大小的各种计算功能单元，包括元素级向量加法、元素级向量乘法、快速数论变换（NTT）以及 CRB 等模块。

第四部分是中间的下半部分，这一部分是输出缓存，输出缓存用 FIFO 缓存。因为 FIFO 缓存非常适用于这个场景，由于 FIFO 缓存采用了先进先出的原则，每次离开队列的元素都是最早进入的元素，因此相比其它的缓存结构，他更好管理，在使用大量数据的时候，使用 FIFO 缓存会使结构更加高效。FIFO 缓存非常简单易懂，没有复杂的逻辑关系和运算，只需要按照队列的顺序来读写数据即可，这使得它易于实现和调试，以便于人们理解他的工作原理。

3.3 本章小结

本章主要进行完全同态加密加速器的介绍，第一小节先是对为什么要用硬件加速器进行了介绍，然后对基于 FPGA 的完全同态加密加速器 F1 加速器的架构以及优化和加速器的方法技巧进行了描述，接着对基于 ASIC 的完全同态加密加速器 Craterlake 加速器的架构以及优化和加速器的方法技巧进行了描述。第二小节根据基于 FPGA 的完全同态加密加速器 F1 加速器和基于 ASIC 的完全同态加密加速器 Craterlake 加速器的架构进行分析和整合，提出了接下来本文将要模拟的加速器，并描述了加速器的各个部分的组成和作用。

4 模拟器设计和实现

4.1 模拟器系统设计框架

4.1.1 模拟器概述

硬件加速器模拟器是一种软件工具，用于模拟硬件加速器的行为和性能。它们可用于开发、部署和测试许多类型的加速器，在不同级别上提供各种功能和复杂度。硬件加速器模拟器在硬件设计过程中发挥着至关重要的作用，不仅可以提高开发效率、缩短上市时间，同时还可以减少硬件故障和成本，并且为下一代加速器的开发奠定了基础。

硬件加速器模拟器的作用具体包括以下几点：开发，硬件加速器模拟器可以用于加速器设计过程中对加速器的全面验证，包括性能、功耗和可靠性等。部署，硬件加速器部署前，使用模拟器进行功能和性能验证可以减少错误和排除缺陷的时间和成本。测试，使用模拟器来测试硬件加速器在不同应用场景下的行为和性能，从而精确评估硬件加速器的性能特征，对后续优化工作具有指导作用。

硬件加速器模拟器主要有以下几种类型：

系统级模拟器：系统级模拟器旨在对整个计算机系统进行全面的模拟。它可以模拟操作系统、应用程序和硬件设备之间的交互过程，从而实现系统功能和性能的评估、验证和优化。系统级模拟器的主要作用有性能评估、可靠性评估。系统验证和计算机架构研究。性能评估，可以对计算机系统的性能进行定量分析，例如测量其处理器和存储介质的速度、带宽和延迟等等。这些数据对于基准测试、设计优化以及性能瓶颈分析非常有用。可靠性评估，可以对系统的可靠性进行分析、确定潜在问题，并查找和排除故障。此外，还可以对系统的容错特性进行测试，确保系统能够成功地恢复到预期状态。系统验证，可以通过模拟不同场景下的运行情况，对系统的完整性和正确性进行验证。这也有助于提前发现分析系统中的错误或问题，并及时纠正。计算机架构研究，可以模拟各种计算机系统，用于研究计算机体系结构的功能、优化以及新技术的设计等方面。

RTL 级别模拟器：RTL 级模拟器(RTL Simulator)是一种硬件描述语言(HDL)仿真工具，通常用于验证集成电路(IC)设计的正确性。它可以将设计描述为使用逻辑电路的门级元素，并在时钟周期级别模拟其行为。RTL 级模拟器有两个主要的应用场景：1.在集成电路的设计和验证阶段中，验证单个功能模块和整个系统的正确性和性能；2.在驱动开发和测试过程中，执行各种信号分析以识别问题或故障点。RTL 级模拟器的主要原理是语法分析和抽象模型生成。简单来说，它通过读取并解释 HDL 代码生成模型，然后对输入注入刺激信号并对输出进行分析。由于 RTL 级模拟器对门级元素的模拟，它们运行速度较快，因此也称为“快速仿真器”。常见的 RTL 级模拟器包括 ModelSim、VCS、Verilator 等等。这些工具通常支持多种硬件描述语言，如 Verilog 和 VHDL，以及 C/C++等高级编程语言。RTL 级模拟器可模拟从复杂的处理芯片到嵌入式微控制器的各种电路，同时还可以直接与电路板（PCB）中的设备集成。它可验证电路的各

种功能，检测和处理时序问题、信号噪声、功耗等特征。RTL 级模拟器在实际工程中发挥着非常重要的作用。它们可以提供一种可靠的方法来开发并进行 IC 设计的验证。在代码编写过程中，使用仿真器可以帮助排除潜在的硬件漏洞，从而节省设计时间，并且加速硬件产品上市速度。同时，这些工具还可以探究 CPU 驱动程序的行为，以便找到性能瓶颈并进行优化。

行为级别模拟器：行为级模拟器（Behavioral Simulator），主要用于逻辑电路的行为级仿真并验证。与 RTL 模拟器不同，行为级模拟器主要关注实现设计功能、处理器性能，并对目标体系结构的管线程序进行高层次的仿真评估。行为级模拟器可以处理高级编程语言描述的设计，并将其转换为一组硬件操作。在真实物理设备中，这些操作会直接影响硬件状态寄存器的值和通过管线流动的其他状态位。相比较 RTL 模拟器， 行为级模拟器更加抽象，它可能使用 C, C++, SystemC 或 SystemVerilog 等高级语言框架来获取行为级别的有效模型。行为级模拟器被应用于 ASIC （集成电路）开发过程的早期阶段. 它们可帮助设计人员快速建立系统构造，评估方案并验证正交功能。无需依赖硬件，只需要仿真验证就能快速确定不同设计选项对 CPU 性能的影响。值得注意的是，行为级模拟器通常也会在 ASIC（集成电路）的拓扑结构变化之后进行大规模系统级的仿真和验证。此时它们在业界被广泛运用，以验证硬件设计的正确性，并判定如何才能发现可能的工程问题。行为级模拟器是在系统设计、集成电路开发与测试中广泛应用的工具。其抽象模型将硬件描述以可读形式呈现，并提供了一种快速有效的方法来验证 CPU 性能、功能和互操作性。在日益增长的 IC 设计领域中，行为级仿真已成为设计开发及验证过程不可缺少的环节。

4.1.2 模拟器系统设计

本文所设计的模拟器是行为级模拟器，主要通过模拟一组同态运算操作在经过从片外高带宽内存读取数据到片上输入 buffer，同时计算核从片上输入 buffer 读取数据就行预算，然后计算核输出运算结果到输入 buffer 留重用或者直接输出到输出 buffer 留返回到片外高带宽存储器中的这一过程来获得加速器架构的参数与加速器的性能指标的关系。

模拟器的实现开发环境如表 4.1 所示：

表 4.1 模拟器设计开发环境

开发环境配置信息
操作系统：Windows10 版本控制与维护：Git 分布式版本控制系统 开发语言及版本： python3.7 编译环境：Pycharm Community Edition 2021 所用到的第三方库：pandas、csv、matplotlib、tqdm、argparse 等等

软件系统框图是一种表示整个软件系统的高层次设计视图，它将整个软件系统以及它们之间的关系、接口和交互等抽象概念以一种可视化的方式呈现出来。通常，软件系统框图以块状图或流程图的形式展现，并封装了一个完整的子系统，指导开发人员进行程序结构设计。如图 4.1 所示，是本文模拟器的软

件系统框图，一部分是左半部分的下一章将进行详细介绍的用于实际实现和分析参数的实验部分，另一部分是模拟器的主体部分，包括用户接口部分、用户配置解析与数据结构定义部分、模拟器的总执行调度部分以及最重要的内存仿真和计算仿真部分。其中内存仿真又包括片外高带宽内存的读和写仿真、片上输入多 buffer 的管理、输入多 buffer 中的单 buffer 的管理以及输出缓存 FIFO 缓存的维护。计算仿真的内容主要是根据要执行的同态操作生成对应的输入输出数据流以及该操作所需的计算延迟用来对作为内存仿真的输入。接下来将对模拟器的五部分的各自的功能以及接口定义进行描述，具体的细节以及实现将放到下一小节。然后再对如何利用这样一个架构获得加速器的性能进行描述。

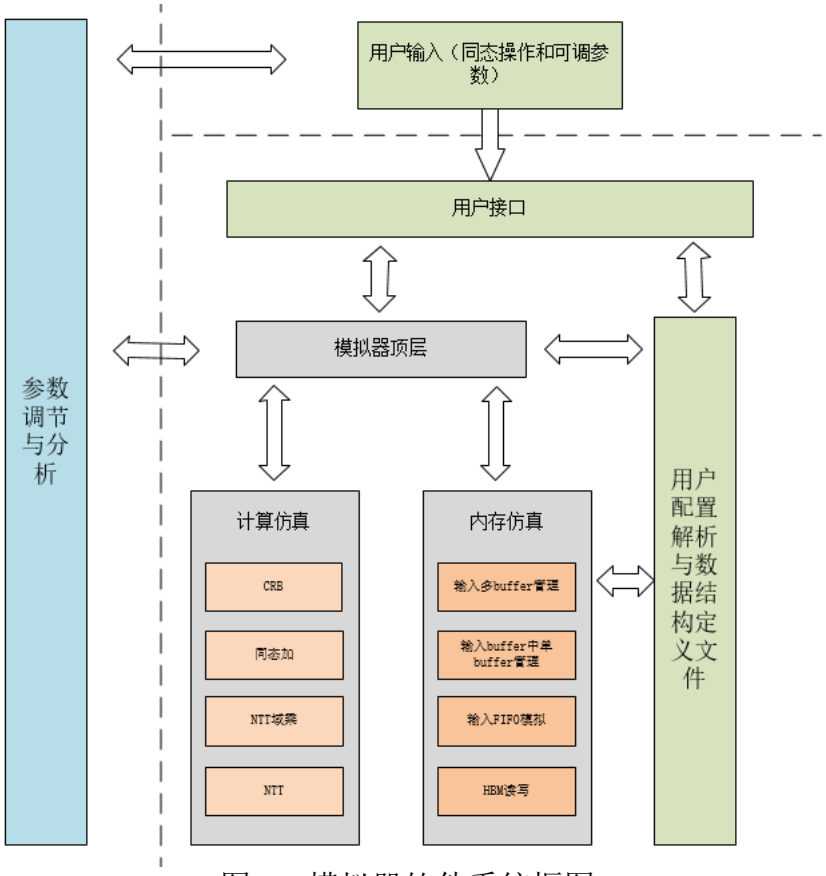

图 4.1 模拟器软件系统框图

（1） 用户接口

用户接口是软件系统和用户之间的一个交互层，它决定着用户如何与软件进行交互并操作软件。用户接口的作用在于让用户能够方便地使用软件功能，以满足其需求和完成任务。用户接口还可以增强软件的易用性、可理解性和用户满意度。

图 4.2 展示了模拟器的输入数据和输出数据，其中输入数据包括加速器的参数配置和加速器输入同态操作流。模拟器通过输入的加速器参数对加速器进行初始化，然后开始根据输入的同态操作数据流顺序执行同态操作，先把数据从片外高带宽内存中取到片上输入缓存汇总，然后计算核开始从片上输入缓

存中读取数据然后执行对应操作，并在执行过程中跟踪记录输入缓存中的数据以及重用队列中的数据以及输出缓存的数据信息以便用来分析从片上缓存中读取数据的读取延迟以及将计算后的数据写回的写回延迟。同时在记录执行总周期时对不同成分的延迟进行分类统计，具体有哪几类以及如何计算将在后面描述。最后将内存中的数据的跟踪信息、各个操作的执行周期数以及总执行周期数的组成输出为三个结果文件。

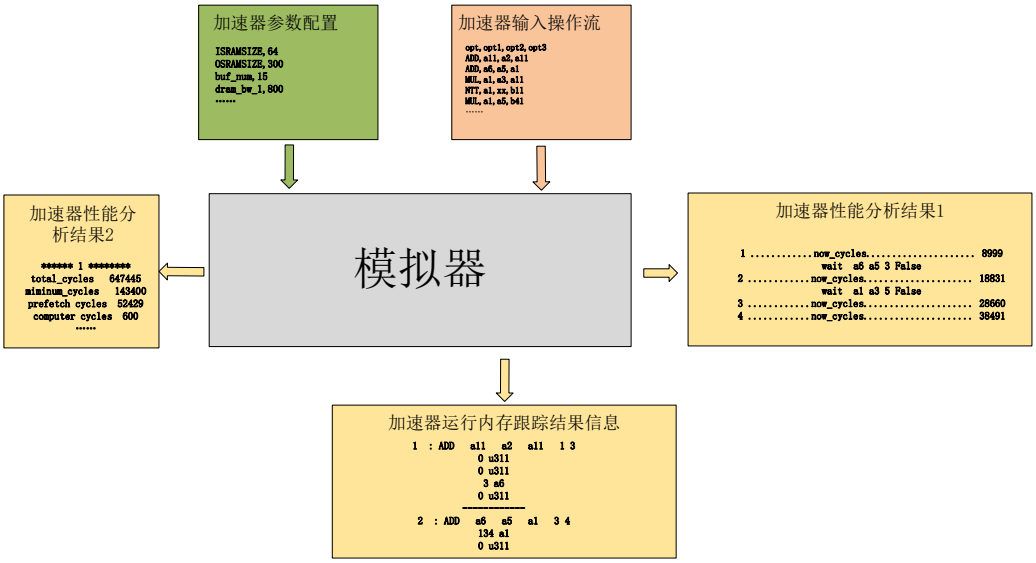

图 4.2 模拟器的输入和输出数据

其中参数信息包括同态加密数据的密文的长度、同态加密密文数据的残差多项式的长度、CKKS 算法所支持的最大乘法深度、算法的安全系数、加速器的片上输出缓存大小、加速器的片上输入缓存的总大小、加速器的片上输入缓存的子 buffer 个数、加速器运行前从片外高带宽内存中预取到输入缓存的密文个数、计算核中各个计算功能单元的计算延迟大小、片外高带宽内存和片上输入缓存之间的输入带宽大小以及输出带宽大小。

输出数据包括三部分内容。第一个是每个同态操作在加速器中所需的执行周期数，也就是图中右边的的加速器性能分析结果 1。第二个是加速器执行所有同态操作的所需的总的执行周期数，以及总周期数的组成情况，也就是左边的加速器性能分析结果 2。该文件的每一条信息有加速器执行所有同态操作总的周期数、加速器理论最小所需执行周期数、加速器运行前从片外高带宽内存中预取数据到片上输入 buffer 中所需的预取周期数、计算核执行输入的同态操作流的所有操作的总的计算核执行周期数、计算核从片上输入缓存中读取数据的时候的读取等待周期数、将计算核的输出数据结果输出缓存 FIFO 写回到片外高带宽内存的写回等待周期数以及所有的输入同态操作都执行玩成之后输出缓存 FIFO buffer 中剩余的数据写回到片外高带宽内存中所需的最终完成周期数七个部分组成。 第三个输出就是在加速器运行过程中对每一个操作执行完成之后片上输入缓存中的数据、片上重用数据的维护信息以及片上输出缓存中的数据的记录。

（2） 用户配置解析与数据结构定义文件

在对模拟过程中，就像上面用户接口部分介绍的那样，我们需要输入两个输入文件,这里我们把这两个文件叫做用户配置文件，那么当这个用户配置文件输入到模拟器中之后，我们肯定是需要对其进行解析获取所需信息然后对加速器进行初始化的。

就像表 4.2 所显示的用户配置解析模块的 python 类的描述那样，该模块由四部分组成，第一个是向模拟器顶层调度模块开放的接口，也就是一些参数和同态操作流。第二个是读取解析参数配置文件功能函数，用来解析参数配置文件。第三个是读取解析同态操作流文件功能函数，用来解析同态操作流文件。第四个是对所有功能模块开放的接口，用于分享一些共有参数（第四部分并没用定义成表 4.2 类中的方法，而是单独定义为了全局共享函数）。

表 4.2 用户配置解析模块类声明

Class **Param_Config** (builtins.object) #声明 Param_Config 类
#类内数据定义，通过__init__()函数，更多具体数据定义不再描述，更具体信息可以参考附录代码 **__ParamData__** 加速器参数 **__OptDate__** 同态操作流数据
#类内方法声明 **Read_param_config_file**(self, config_file_path) 该方法用于读取解析用户参数配置文件 ：config file path：参数配置文件路径 **Read_opt_config_file**(self, config_file_path) 该方法用于读取解析同态操作流文件 ：config file path：操作流文件路径

在配置文件的输入形式上，除了直接定义一个文件以外，我们也支持在运行主程序时通过指定参数的形式进行参数配置信息的指定。除了参数配置信息的读取和同态操作流的读取之外，用户配置文件解析对加速器模拟的模拟器顶层调度开放一些接口，以便模拟器顶层模块可以方便的获取想知道的加速器信息以及同态操作流。

具体如何从配置文件和同态操作流文件中获取信息以及对模拟器顶层模块开放哪些信息都将在下一小节模拟器的具体实现中进行详细介绍。

（3） 模拟器顶层调度

如表 4.3 所示是这个模块的类实现。这个部分起到整体协调作用，向上可以从用户配置解析模块获得想要的用户配置信息以及同态操作流信息，向下可以综合调用计算仿真模块和内存仿真模块。主要流程就是先从用户配置解析文件获得配置信息进行初始化然后依次循环执行同态操作流中的操作，先根据操作类别从计算核模块获取输入数据流信息和输出数据流信息，然后根据输入

数据流信息调用内存仿真模块模拟从片上输入 buffer 中获取该操作对应的数据流，最后再根据输出数据流信息调用内存仿真模块模拟将输出数据通过输出 buffer 写回片外高带宽内存，如果该输出数据需要重用，则同时写回到片上输入 buffer 中留到接下来重用。 其具体的执行流程如图 4.3 所示。

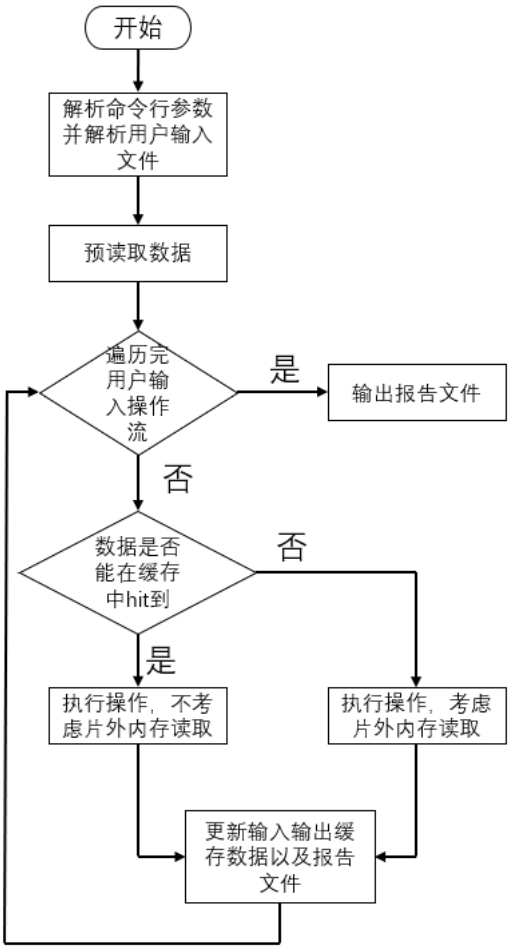

图 4.3 模拟器顶层模块总流程图

表 4.3 模拟器顶层调度模块类声明

Class **Simulator**()　#声明 Simulator 类，这个模块用于总体调度
#类内数据定义，通过__init__()函数，更多具体数据定义不再描述，更具体信息可以参考附录代码 **__UpData__** 　　从上层获取的参数 **__Cycles__** 　　各类型周期数定义
#类内方法声明 **ArgumentParser** ( ) 　　用于获取并解析命令行中指定参数 **GetInformation** (Upclass) 　　获得用户配置文件解析模块中需要传递下来的数据

：Upclass：在本模块中定义的用户配置模块的类

Run ( )

整体运行框架，先预取数据，然后根据得到的操作流依次调用执行函数执行，然后更新输入输出 buffer，并记录输入输出 buffer 的存储数据信息。

Run_Opt (opt, start_cycles) - > end_cycles

模拟执行 opt 操作（会根据此时输入 buffer 中数据存储情况分类操作），先从根据操作类型调用计算仿真模块获取数据流，然后调用内存仿真模块读取输入数据流，再调用内存仿真模块写回输出数据流。并根据开始周期获得结束周期时间，同时得到输入输出 buffer 的等待时间。

：opt：操作类型及数据 id 信息，根据数据 id 信息可以得到其存储地址

：start_cycles：是一个结构体，包含各种操作周期在执行这条操作之前的数值

：end_cycles：与 start cycles 对应，返回的是在执行了这条操作之后的。

（4） 计算仿真

计算仿真模块主要是模拟加速器的计算核的功能单元，通过输入对应操作的类型，可以得到操作的输入数据流和输出数据流以及操作延迟信息，由于完全同态加密加速器的设计中所有的计算核的功能单元的设计都是采用固定输入输出大小且固定单元延迟的设计方法设计的，所以我们要向得到输入输出数据流和功能延迟在这里还是很容易的。具体的类声明如表 4.4 所示。

表 4.4 计算核仿真类定义

Class **computer**（） #声明 computer 类
#类内方法声明 **Get_InputStream** (operation) 根据输入的 operation 返回对应的输入数据流，在本文的实现中是返回具体的存储起始地址与数据大小。 ：operation：要获取的操作的具体信息，包括 id、类型等等，用于定位操作。 **Get_Latency** (operation) 根据 operation 中的操作的类型来返回对应操作的所需周期数，该数值来自于用户指定，以方便调节。

（5） 内存仿真

内存仿真是本文模拟器仿真的重点内容，整体上来讲就是仿真片上输入多 buffer 结构的缓存从片外高带宽内存读取数据、计算核从片上输入缓存读取数据、计算核输出数据写回片上输入缓存、计算核输出数据写入输出 FIFO 缓存以及片上输出 FIFO 缓存写回片外高带宽内存这五个部分的内容。

在具体的实现方法上，我们用两个类来进行仿真，分别仿真输入缓存和输

出缓存，同时用两个队列来分别维护 hit 信息和重用信息。输入缓存的的数据定义和方法声明如表 4.5 所示，输出缓存的方法声明类似，不过输出缓存和输入缓存的结构不一样，所以具体模拟方法不同，这将在下一小节具体描述。

表 4.5 输入缓存控制器类定义

Class **Controller**() #输入 buffer 的控制器，用于模拟输入 buffer
#类内数据定义， 此处只描述几个重要的数据，其它数据如需了解可查看附录代码。 **__ReusedDeque__** 一个双端队列，表示目前还在片上重用的操作数，方便维护踢出发生内存存储冲突时还在最早的重用操作数。 **__SurplusQueue__** 单端队列，用于维护 buffer 中剩余的空闲 buffer **__OptSramMap__** 是一个字典，键值是操作数名，值是输入 buffer 的索引，用于判断操作数是否能在片上输入缓存中 hit 到。
#类内方法定义 这里只介绍一些从输入缓存中读取数据的方法，写入的方法类似，具体信息可以参考附录代码 **Set_read_buf_prefetch** (self, num) 用于在开始运行加速器之前从片外高带宽内存中预取数据到片上输入缓存。 ：num：预读取的 buffer 量 **Update_Buffer** (self, cycles) 用于更新输入缓存转态信息以及存储数据信息到当前周期。 ：cycles：表示当前模拟到第几个执行周期 **Get_next_opt** (self, now_id) 用于模拟从输入缓存中获取 now id 这个操作数数据到计算核中。 : now_id：表示要读取的操作的操作数的起始地址

4.2 模拟器的子模块实现

在上一小节中我们详细的介绍了模拟器的整体系统框架以及用户配置文件解析模块、计算核模拟、内存模拟以及模拟器顶层调度模块这几个部分的实现所用到的部分数据以及部分功能函数的声明。在这一小节中将更具体和详细的介绍模拟器的每一个子模块是怎么实现的，流程图是什么样的，以及每一个子模块中的每一个功能函数的实现。

还是按照上一小节中的图 4.1 所显示的那样，将模块的划分分为用户配置文件解析模块、模拟器顶层调度模块、内存模拟模块以及计算核模拟模块四个部分分别进行介绍。

（1） 模拟器顶层调度模块

该模块的功能可以完整的显示模拟器的工作流程，其整体流程图如图 4.3 所示，首先是读取命令行参数并调用用户配置文件解析模块进行对用户输入的信息进行解析，然后初始化加速器，接着开始根据从用户输入文件中得到的操作

流信息调用操作执行函数模拟执行操作，最后根据执行结果更新执行周期信息和输入输出缓存的转态和数据。

表 4.6 是具体的在如何模拟加速器执行一个操作的伪代码，该操作的所需操作数都是确保已经在输入缓存中的。

表 4.6 模拟执行一条操作指令的伪代码

函数名称	Run_once()
输入参数	操作类型 optclass、两个源操作数和目的操作数的起始地址 opt1 opt2 cnt 、开始周期结构体 start_cycles
输出参数	结束周期结构体
算法主体	获取操作 optclass 的延迟 cmt_latency； 更新输出缓存的状态到 cmt_latency + start_cycles 周期； 获取操作 optclass 的操作的输入输出流矩阵； 如果 cnt 是零： 先读取 cmt_latency 周期的数据使计算核开始工作； 将输出数据流矩阵写回输出 buffer 直到输入数据全部进入计算核； 否则： 如果目的操作数在输入缓存中或者目的操作数不会被重用： 将输出数据流写回输出缓存； 否则： 将输出数据流写回输入缓存和输出缓存

表 4.6 中的这个伪代码是在操作数确保在输入缓存中时候的执行流程，那么如果在要模拟这个操作的执行的时候发现这个操作不在输入多 buffer 缓存中怎么办呢，此时就需要先确保要执行的这个操作的操作数在输入缓存中存的数据量足够片上带宽大小之后再开始将数据读取到计算核，然后这个时候需要再次不断的循环判断这个操作数是否在输入缓存中有足够这个周期使用的数据量，够的话就执行，不够就等待片上输入缓存从片外高带宽内存中获取到足够多的数据之后再执行。其流程图如图 4.4 所示，首先感谢输出缓存 FIFO buffer 到当前执行周期，防止下面出现冲突，然后和上面一样，先根据操作种类获取输入输出数据流，不过这里不用获取操作延迟，因为这里是出现 stall 的情况，然后循环判断输入数据流是否都已经输入到计算核，如果是，就输出最终报告文件，如果没有，就先判断输入缓存中的数据是否足够这一次进入计算核的数据量，如果足够就直接取出然后执行再写回 FIFO，如果不够就先等输入缓存从片外高带宽内存获得足够的数据之后再执行然后写回 FIFO，并且要及时更新 FIFO 状态（写回片外高带宽内存）。这里不用考虑目的操作数是不是要片上重用，因为此时片上输入缓存数据不足，此时写回输入缓存效益不大。

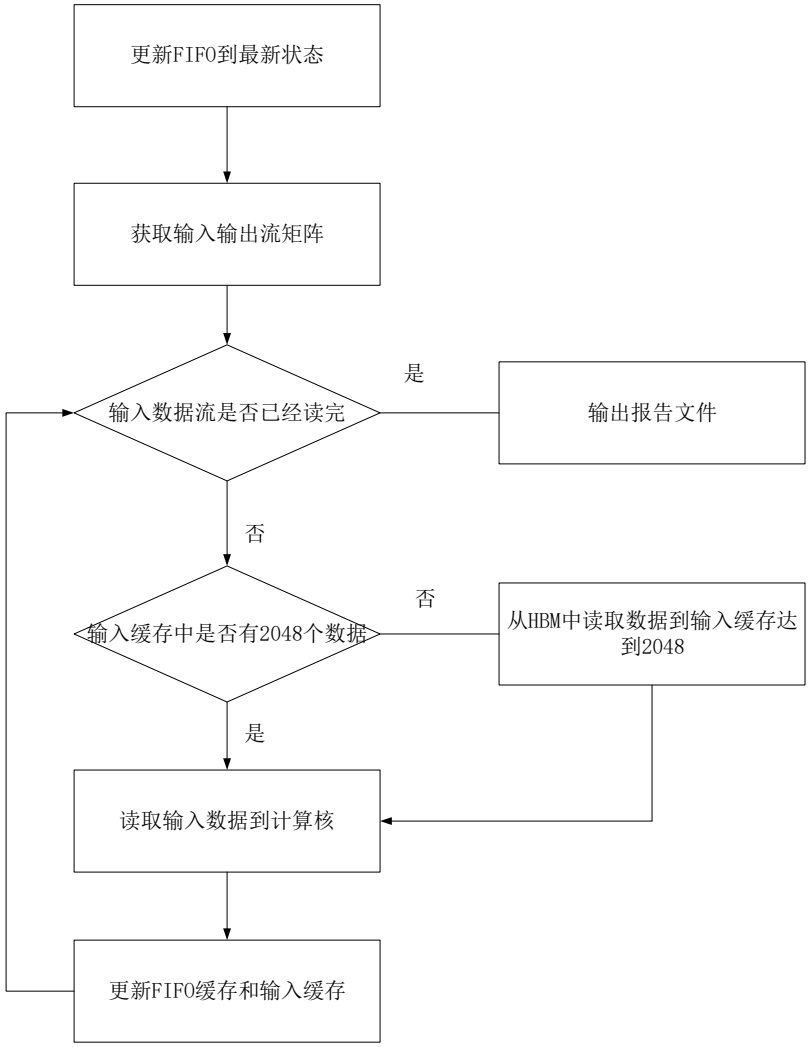

图 4.4 执行一次操作的流程图（没有 hit 到数据情况）

（2） 用户配置文件解析模块

这一部分包括两块内容，一个是向上获取用户的配置文件信息，一个是对外传递数据。其中读取用户配置文件信息又分为读取的是参数配置文件还是操作流文件。

参数配置文件的存储形式是如表 4.7 所示，所以只需要将该文件存成 csv 文件，然后用 panda 库来读取成 DataFram 格式，然后获取其中数据就可以了。

表 4.7 参数配置文件存储信息

Param	Val
N	65536
L	50
I	60
INSRAMSIZE	64
OUTSRAMSIZE	300
BUFNUMBER	4
DRAMBANDW1	800
DRAMBANDW2	400

PREFETCHNUM	4
COMPUTERLATENCY	60

操作流配置文件的存储是如表 4.8 所示的那样，所以在读取该配置文件的时候，也是先用 panda 库读取一下这个文件为 DataFrame 类型的数据，然后循环遍历这个操作流，如果是 NTT 操作的话说明只有一个源操作数，此时就只需要保存一个源操作数到操作数链表中，如果不是 NTT 操作，则需要保存两个操作到操作数链表中，同时所有操作的目的操作数都保存到另一个操作数链表中，然后先通过源操作数链表找到需要重用的数的队列，具体操作为从后往前判断当前操作是够已经出现过，如果出现过就说明需要重用，则把这个操作数加入需要重用的映射中，找到重用映射之后，再把两个操作数队列合到一起，留之后使用。

表 4.8 操作流文件存储信息

Optclass	Opt1	Opt2	Opt3
ADD	A11	A2	A11
ADD	A5	A6	A21
MUL	A1	A11	A3
NTT	A5	XX	A1

（3） 计算核仿真

计算核的仿真在本文的实现中是比较简单的，因为计算核功能单元的所有操作都是完全流水的，都是用于固定延迟的，都是不会出现 stall 现象的，当然片上出入 buffer 是有可能出现没有 hit 到操作数的情况的，所以这个完全流水并不是全周期的完全流水。所以在计算核的几个方法函数中，第一个返回计算延迟，只需要先根据模拟器顶层文件从用户文件配置解析模块得到的参数来进行配置，然后提供返回函数，在外界需要的时候，直接返回就可以了。第二个是返回输入数据流矩阵，这个矩阵的大小是操作数的大小，每一行的长度是片上带宽的大小，也就是每一次需要输入到计算核的数据的个数，那么行数也就可以对应确定了，而且这里由于我们把没有 Hit 到的情况放到了模块顶层调度模块去考虑了，所以这里的返回输入数据流和输出数据流也可以直接返回矩阵的行数来代替。如果想把没有 Hit 到的情况放到这里考虑，就需要返回矩阵，等待输入缓存获取数据的那些周期，矩阵中对应的行就是可以都给零或者是负一之类的特殊数据。

（4）内存仿真

就像上一小节说的那样，内存仿真部分是本文模拟器的重点，它实现了对操作数读出和写入的跟踪，得到了跟踪矩阵，可以以此来获得总的执行周期，以及各部分执行周期。

内存仿真中有很多功能函数，这里就只对计算核读取数据和写回数据这两

个与外部相连的使用进行介绍，更多的功能函数可以查看附录代码。下面介绍一下计算核是如何从输入缓存中读取数据的以及计算核的输出数据是如何写入的输出缓存 FIFO 的。

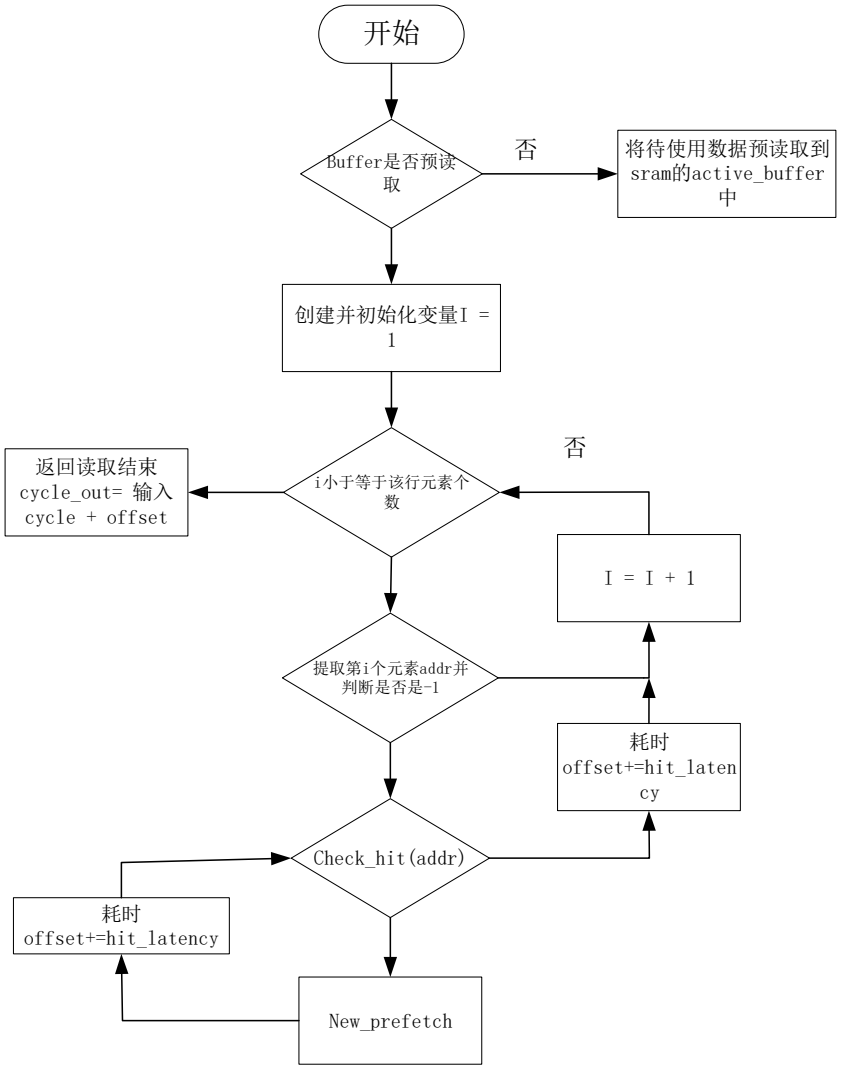

图 4.5 读取输入缓存数据流程图

如图 4.5 所示是关于计算核如何从片上输入缓存中读取数据的流程图（双 buffer 结构，我们使用的是多 buffer 结构，操作稍微复杂一点，但是类似），首先要判断输入缓存中是否已经预读取过数据，如果没有的话，就先将待使用的数据预读取到输入缓存中，然后循环判断数据是否已经全部取出，如果没有的话，就读取一行，在读取之前，要先判断在输入缓存可读取范围中是否能 hit 到数据，如果不能的话就先等到预读取满了数据之后再读取。

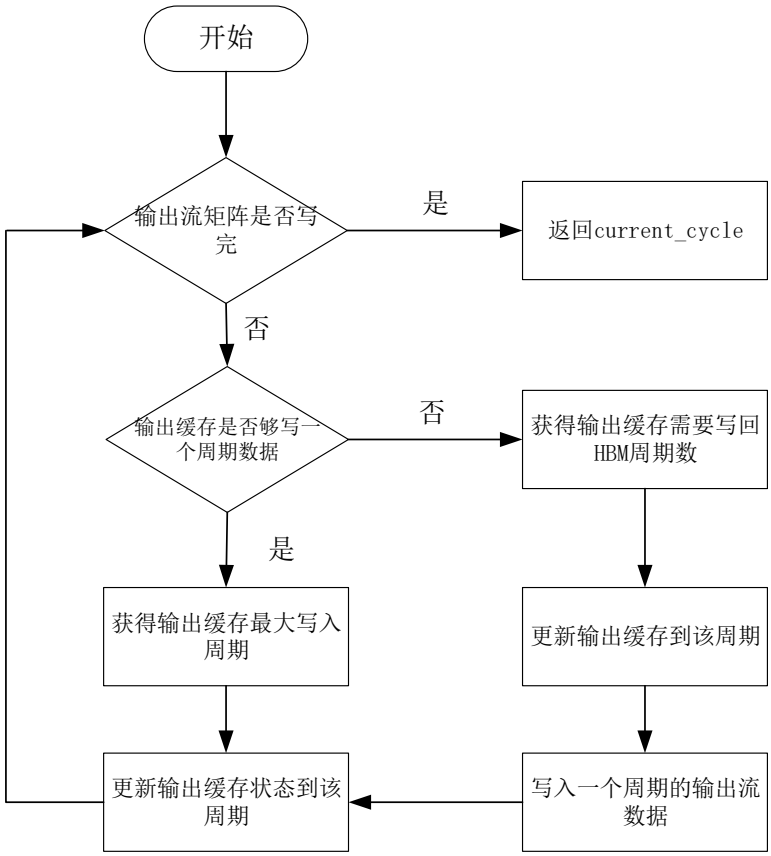

图 4.6 输出数据流写入 FIFO 缓存

如图 4.6 所示描述了一个输出流矩阵是如何写入到输出缓存 FIFO 的，循环判断输出数据流是否已经全部写完，如果全部写完了就返回当前写完的周期数以及 FIFO 缓存的状态信息，如果没有写完就先判断输出缓存 FIFO 的剩余空间是否足够写入一个周期的（也就是片上带宽大小）数据量，如果不够的话，就要先等待一些时间让输出缓存空间中的数据写回片外高带宽内存中一部分，所以要先获得使片上输出缓存拥有足够剩余空间所需要的的周期数，然后更新输出缓存到这个周期，此时输出缓存的剩余空间就足够写入一个周期的输出流数据，然后更新输出缓存到这个周期，而如果输出缓存剩余空间足够大的话，就先获得最多写入周期数，然后直接更新到该周期。

4.3 本章小结

本章的目的是为了详细的介绍模拟器是如何实现的，第一小节介绍了模拟器的整体软件框架以及各个子模块的定位与功能以及相互连接，第二小节介绍了模拟器的各个子模块的一些具体的功能函数的实现以及流程图或者伪代码。

5 模拟器的实验设计和结果分析

5.1 实验设计

实验的最终目的是找到加速器的各个参数与加速器性能之间的关系，从而找到能够以尽可能少的资源获得尽可能好的性能的参数。需要调整加速器参数有密文的长度 N、加密算法允许的最大乘法深度 L、残差多项式的长度 I、安全系数λ、 输入缓存 buffer 数、预读取数、输入带宽、输出带宽、片上带宽等等近十个参数，那么多的参数如何合理的寻找参数与加速器性能之间的关系显然是一个很麻烦的事情。

由于参数太多，所以不可能一张图或者一个实验就得到所有的结果和关系，所以如图 5.1 所示，我们把实验分成 n 组，每一组可以探索 1 到 4 个参数与加速器性能以及加速器执行总周期组成之间的关系，为了能够直观的显示最终的结果，我们最后会使用 matplotlib 对参数和结果进行绘图进行显示。首先就是选定要调整的参数，然后设定好可调范围，循环遍历参数，然后通过命令行对模拟器参数进行指定，再调用加速器执行操作流，得到操作流中每个操作的执行周期数和总执行周期数以及总周期的组成，然后调用响应的绘图功能函数进行绘图。那么整体执行复杂度就是$O\left(\prod_{i=1}^{m} A_i * O(t)\right)$, 其中$O(t)$是模拟器执行一次全部操作流的复杂度，$A_i$是第 i 个参数的范围大小，m 是要分析的参数个数。

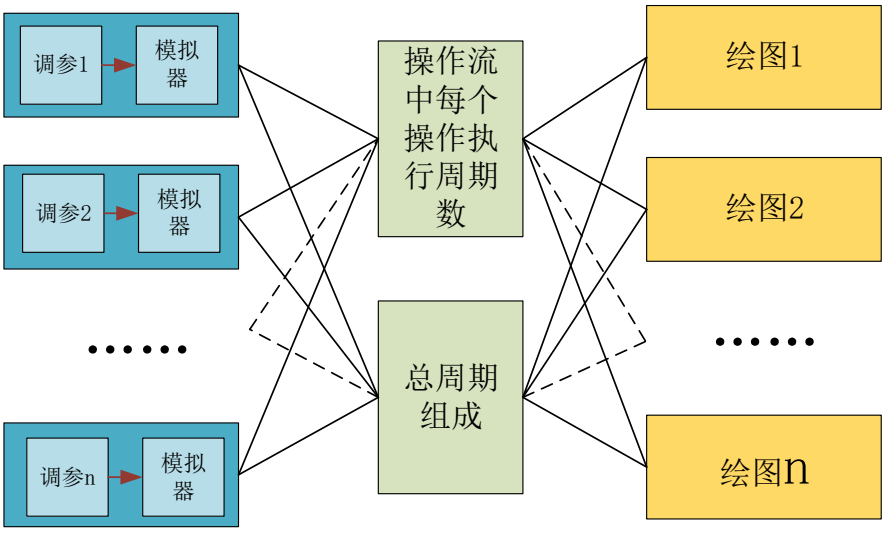

图 5.1 实验设计图

虽然每个实验只用分析 1 到 4 个参数，所以在时间复杂度上是可以接受的，但是由于要调节的参数太多，所以如果想要找到所有参数与结果的关系，就需要几十万次的实验，显然这无论是在时间复杂度还是人力分析结果可实现上都是不现实的，所以我们在使用前必须要确定到底要分析哪些参数，进一步减少要分析的参数个数，以及实现分析一下哪些参数之间的影响更大。

比如在某个场景中，密文的长度就是 65536，是一直恒定不变的，是不需要调整的，那么就不需要分析这个参数，只用一直指定一个定值就可以了。那么

在接下来的实现中，我们将类似的指定一个场景，在这个场景中软件层面我们无法调整且片上带宽很大，主要时间消耗在于片外的交互上，那么这个时候只需要调整片外内存与片上输入缓存的带宽、输出缓存与片外高带宽内存的带宽、片上输入缓存的 buffer 个数、预取 buffer 数以及输出缓存的大小。并且其中输入缓存与片外带宽以及输出缓存与片外带宽的总大小是固定的。那么根据这个，我们设计进行以下几个实验：实验 1：片外带宽以及输出 buffer 大小与总性能的关系、实验 2：片外带宽大小与延迟组成的关系、实验 3 输入缓存 buffer 个数、预取数、输出缓存与总性能之间的关系、实验 4 输入缓存的 buffer 个数以及片外带宽与总性能之间的关系。

片外带宽以及输出 buffer 大小与总性能的关系实验可以大概找到比较合适的带宽大小以及输出 buffer 大概大小，片外带宽大小与延迟组成的关系实验是因为可以结合实验 1 结果看看输出片外带宽的大小的改变具体是影响的哪一块,，实验 3 是为了探索输入输出缓存以及片外带宽与总性能之间的关系，实验 4 类似实验 1 的探索，不过这个是探索输入缓存与带宽的整体影响。

5.2 结果分析

如上一小节所描述的，我们进行了五个实验，接下来将分别分析这五个实验的结果。这里需要注意的是，实验的结果会极大的收操作流的影响，操作流来自于编译器，它决定了加速器的片上重用程度，片上重用程度越高，性能越好，而且影响很大。本文的所有实验都是针对同一操作流，不考虑操作流的改变对加速器参数敏感性的影响。

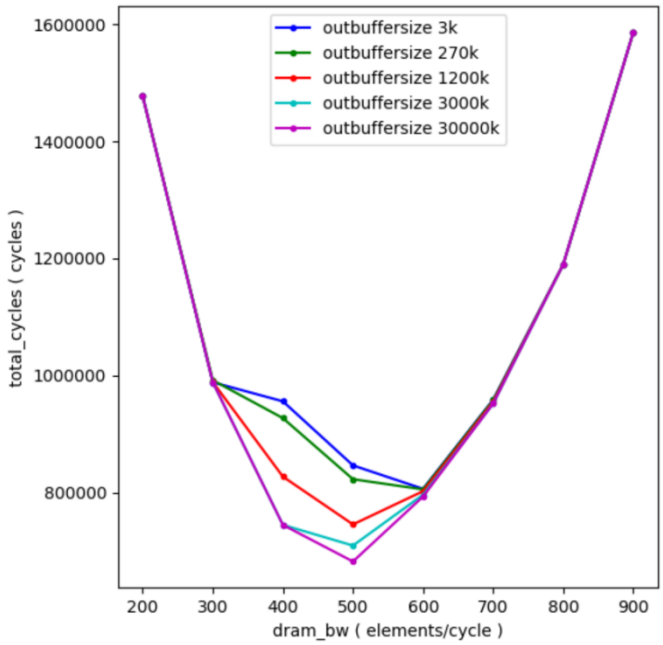

图 5.2 片外带宽以及输出缓存大小与总性能关系图

如图 5.2 所示的片外高带宽内存与片上输入缓存之间的带宽大小以及输出缓存大小与总性能之间的关系，由于总的片外带宽大小是固定的，所以当片外内存与输入缓存之间的带宽大概处于中间时（600），整体性能较好，具体原因我

们在实验 3 中可以看到，并且输出缓存的大小是越大越好的，这也符合我们的直观印象，但是具体为什么输出缓存的大小越大越好可以在实验 2 的分析中看到。基于此，我们在后面的实验中，如果需要指定带宽大小，显然我们就可以取其为 500 左右的数值。

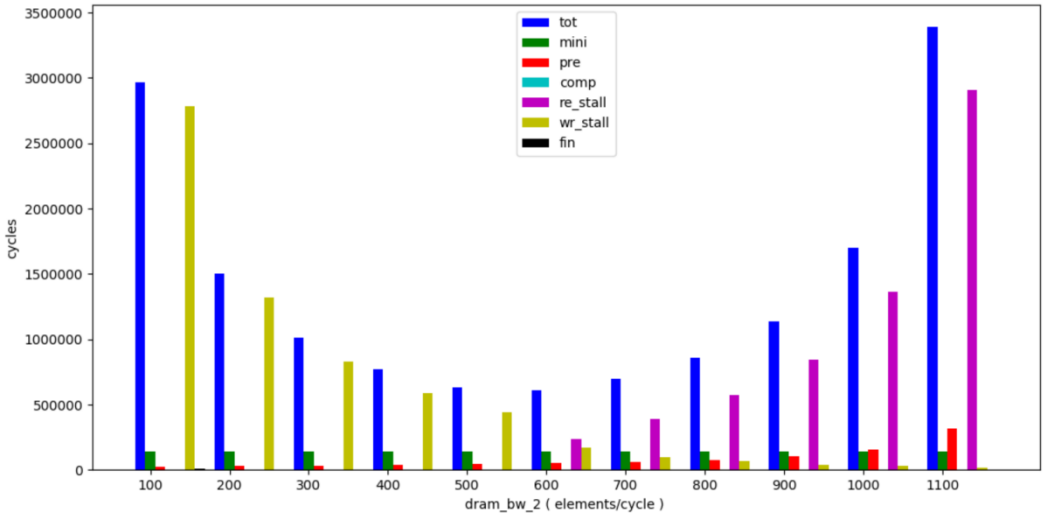

图 5.3 片外带宽大小与延迟组成的关系

实验 2 的结果如图 5.3 所示，结合实验一的片外内存与输入缓存之间的带宽接近中间值，或更确切的说接近 600 元素每周期的时候，加速器总的延迟最小，性能最好，可以看出是因为当带宽变大，也就是图 5.3 上的带宽变小的时候，的时候由于输出缓存到片外内存的带宽小，所以 FIFO 的数据很难写回，那么计算核的生成数据就不得不等待 FIFO 缓存的数据写回，也就是图中你的黄色部分不断增大，导致总体性能降低，当带宽变小，也即是图 5.3 上的带宽变大的时候，相对应的，输入缓存中的数据量不够计算核用的，所以就导致需要等待输入缓存从片外内存中取数据，也就是图中的紫色部分逐渐增大，导致性能降低。

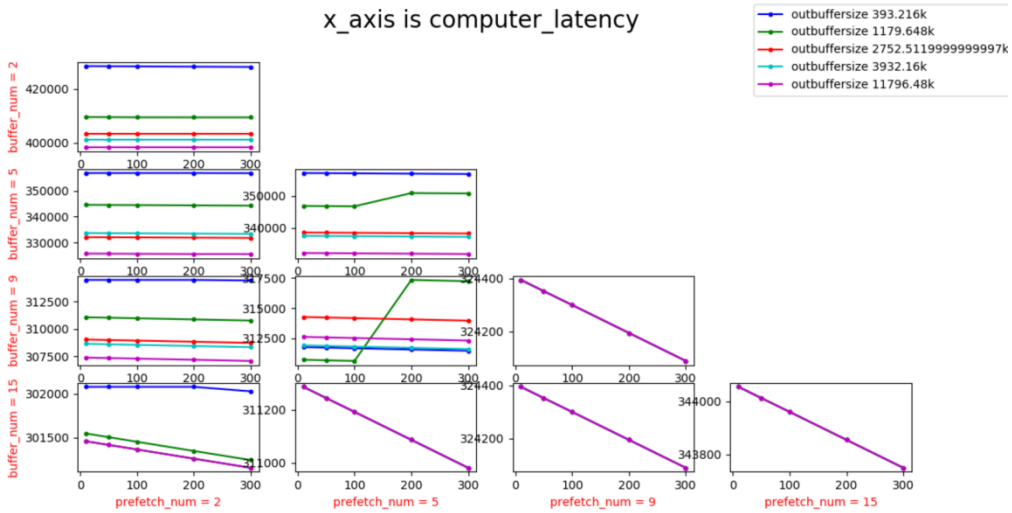

图 5.4 片上缓存的三个参数以及片外带宽与性能关系图

图 5.4 是片上输入缓存的 buffer 个数、加速器运行前要预取到片上缓存中的密文个数、片上输出 FIFO 缓存的大小以及片上输入缓存与片外内存的带宽这四个参数与加速器执行操作流的总周期之间的关系，这个图可以结合其他实验仪器分析，在本图中可以看出片上输入缓存的 buffer 个数以及输出缓存的大小是越大越好的，这与我们的直观理解是相同的，但是随着结果增大，带来的性能改善越来越小，我们将在实验 4 的结果图 5.5 中看到具体情况。预取数和带宽则受其他参数影响。

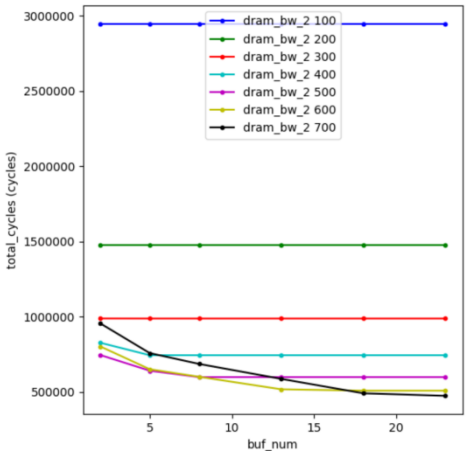

图 5.5 输入缓存 buffer 个数以及片外带宽与总性能之间的关系图

实验 4 的结果如图 5.5 所示，反应了输入缓存 buffer 个数以及片外带宽与总性能之间的关系图，可以看出在带宽取总性能比较好的 500 附件的值的时候，输入缓存的 buffer 数都可以找到一个最小的且总性能最好的点，那么这个点就是我们想要的资源消耗少且性能好的点。

所以在我们假定的这个场景下，通过这四个实验我们就可以发现，当片上输入缓存与片外内存之间的带宽取值 500、片上输入缓存 buffer 数取值 9、预取数为 5 以及片上输出缓存大小取值 1200k 的时候总体资源消耗少且性能好。

5.3 本章小结

本章在基于上一章的基础上介绍了如何获得参数与加速器性能的关系，并进行实验思路和框架分析，然后针对一个假定场景进行了具体的实验设计，并对结果图进行了分析。

6 结论

本文针对现有的基于 FPGA 和 ASIC 的完全同态加密硬件加速器的架构特点设计了专用于完全同态加密加速器的行为模拟器，该模拟器可以根据输入的加速器参数和同态操作流得到加速器架构在这组参数下的三个结果文件，分别是执行每一个操作所需的时间、执行完所有操作后的总执行时间的组成情况以及整个执行过程中输入缓存和输出缓存的跟踪矩阵。

并且对如何在加速器参数很多的情况下进行调整参数给出了建议，以及试验设计架构。同时还举了一个在指定场景下的如何进行实验，以及如何根据实验结果进行参数选择的例子。

本文的主要创新点有两点，一个是设计了一款针对全同态加密硬件加速器的专用行为模拟器用于辅助加速器架构的参数选择。第二个是给出了如何在加速器参数很多时进行实验，并且提供了实验框图。

不过虽然本文目前已经可以辅助加速器架构的参数选取，但是还存在两点问题，一个是本文的模拟器虽然可以得到三个结果文件，但是事实上这三个结果都是更关注于性能，但我们知道架构的设计不能仅仅关注性能，资源消耗和功耗等也都是很重要的因素，而在本文中这些因素需要在选择参数时自己考虑。第二个是本文的模拟器缺乏灵活性，只是针对一个特定的架构的，虽然各种加速器的架构差别不是很大，但是现在加速器架构还在探索中，所以应该让模拟器的内存结构模拟和计算核模拟部分有更多的灵活性和选择性。

接下来的工作可以针对这两点问题进一步完善这个模拟器，第一个问题可以在模拟操作的执行中考虑上资源的总消耗和利用率，功耗的问题可以想办法和其它现有的功耗模拟器想结合。第二个问题可以对模拟器的整体架构进一步扩充，已实现更高的灵活性。

致谢

时光如梭，本科阶段的学习生活即将画上句号。在这四年里，我掌握了专业知识与技能，开阔了眼界。

首先十分感谢范文兵老师和赵路坦老师，作为我的指导导师，给予了我许多科研方面的指导，让我具备了基本的科研素养与能力。-

最后，感恩父母一直以来对我的培养、付出与支持，感谢同学们的陪伴，你们让我的大学四年生活更加充实与美好。

参考文献

[1] 李习习,胡业周.基于全同态加密的安全多方计算探讨[J].电脑知识与技术,2020,16(21):19-22.

[2] 吴昱昊.基于 FPGA 的目标检测系统与加速器设计[J].现代信息科技,2023,7(07):101-104.

[3] Axel Feldmann , Nikola Samardzic , Aleksandar Krastev , Srini Devadas , Ron Dreslinski , Christopher Peikert , Daniel Sanchez. F1: A Fast and Programmable Accelerator for Fully Homomorphic Encryption. IEEE Micro, 2021, 41(1), 42–49.

[4] 黄沛昱,赵强,李煜龙.基于 FPGA 的卷积神经网络硬件加速器设计[J].计算机应用与软件,2023,40(03):38-44.

[5] NVIDIA 联合微软发布行业标准级超大规模 GPU 加速器,助推人工智能云计算发展[J].智能制造,2017,(03):6.

[6] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Nathan Manohar, Nicholas Genise, Srinivas Devadas, Karim Eldefrawy, Chris Peikert, Daniel Sanchez. CraterLake: A Hardware Accelerator for Efficient Unbounded Computation on Encrypted Data. ISCA, 2022.

[7] Yinghao Yang, Huaizhi Zhang, Shengyu Fan, Hang Lu, Mingzhe Zhang, Xiaowei Li. Poseidon: Practical Homomorphic Encryption Accelerator. 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).

[8] Rashmi Agrawal, Leo de Castro, Guowei Yang, Chiraag Juvekar, Rabia Yazicigil, Anantha Chandrakasan, Vinod Vaikuntanathan, Ajay Joshi. FAB: An FPGA-based Accelerator for Bootstrappable Fully Homomorphic Encryption. 2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA).

[9] Shafagh, H., & Dubois-Ferriere, H. (2019). An Energy-Efficient Implementation of Fully Homomorphic Encryption for Resource-Constrained Networks. Proceedings of the 2019 International Conference on Embedded Networked Sensor Systems (SenSys '19), 269–270.

[10] Ananda Samajdar, Jan Moritz Joseph, Yuhao Zhu, Paul Whatmough, Matthew Mattina, Tushar Krishna. A Systematic Methodology for Characterizing Scalability of DNN Accelerators using SCALE-Sim. 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS).

[11] Meinerzhagen, P., Hochberger, C., & Burg, A. (2012). DRAMSim2: A Cycle Accurate Memory System Simulator. Proceedings of the Fourth Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-4), 21-30.
[12] Binkert, N., Beckmann, B., Black, G., Reinhardt, S. K., Saidi, A., Basu, A., Hestness, J., Hower, D. R., Krishna, T., Sardashti, S., Sellier, M., Tata, S., Tavana, M., Vashishtha, V., & Waterman, A. (2011). The gem5 simulator. ACM SIGARCH Computer Architecture News, 39(2), 1-7.
[13] 朱宗武, 黄汝维. 基于高效全同态加密的安全多方计算协议[J]. 计算机科学, 2022, 49(11):345-350.
[14] Wang, X., & Jia, C. (2018). Survey on homomorphic encryption and address of new trend. Journal of Beijing University of Posts and Telecommunications, 41(3), 39-48.
[15] 苏昀暄, 涂正, 王绪安, 林川. 一种基于 CKKS 同态加密与神经网络的安全人脸识别方案[J]. 兰州理工大学学报, 2023, 49(02):103-109.
[16] Wang, G., Zhang, X., & Zhan, J. (2019). A survey on homomorphic encryption schemes: Theory and implementation. Journal of Information Security and Applications, 48, 102354.
[17] 刘钦菊, 路献辉, 李杰, 王鲲鹏. 全同态加密自举技术的研究现状及发展趋势[J]. 密码学报, 2021, 8(05):795-807.